JVM under the hood

Brijesh Nekkare

Important Disclaimers

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.

WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.

ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.

ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.

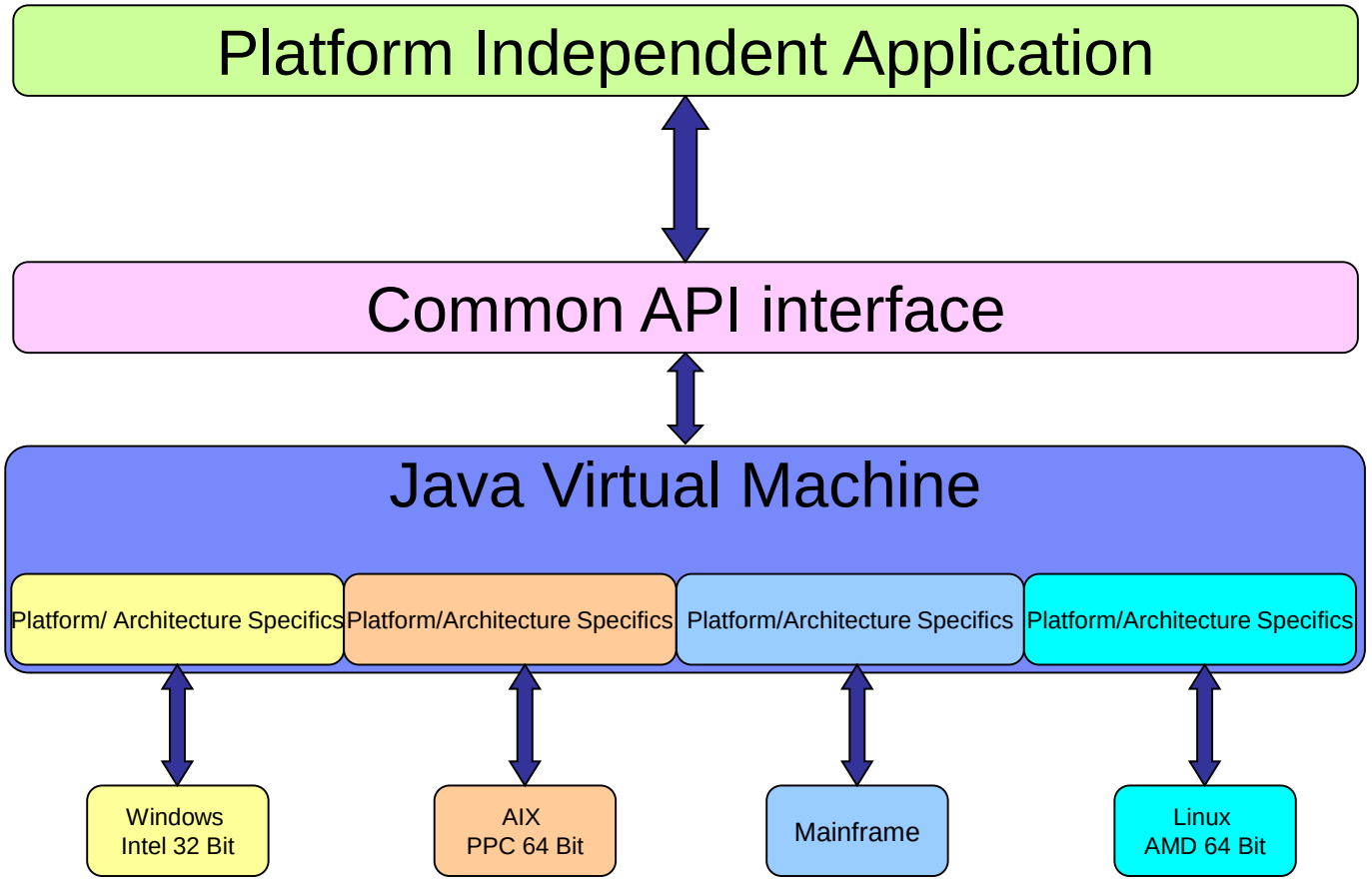
IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM’S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.

IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.

NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:

- CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS

Java Runtime Environment



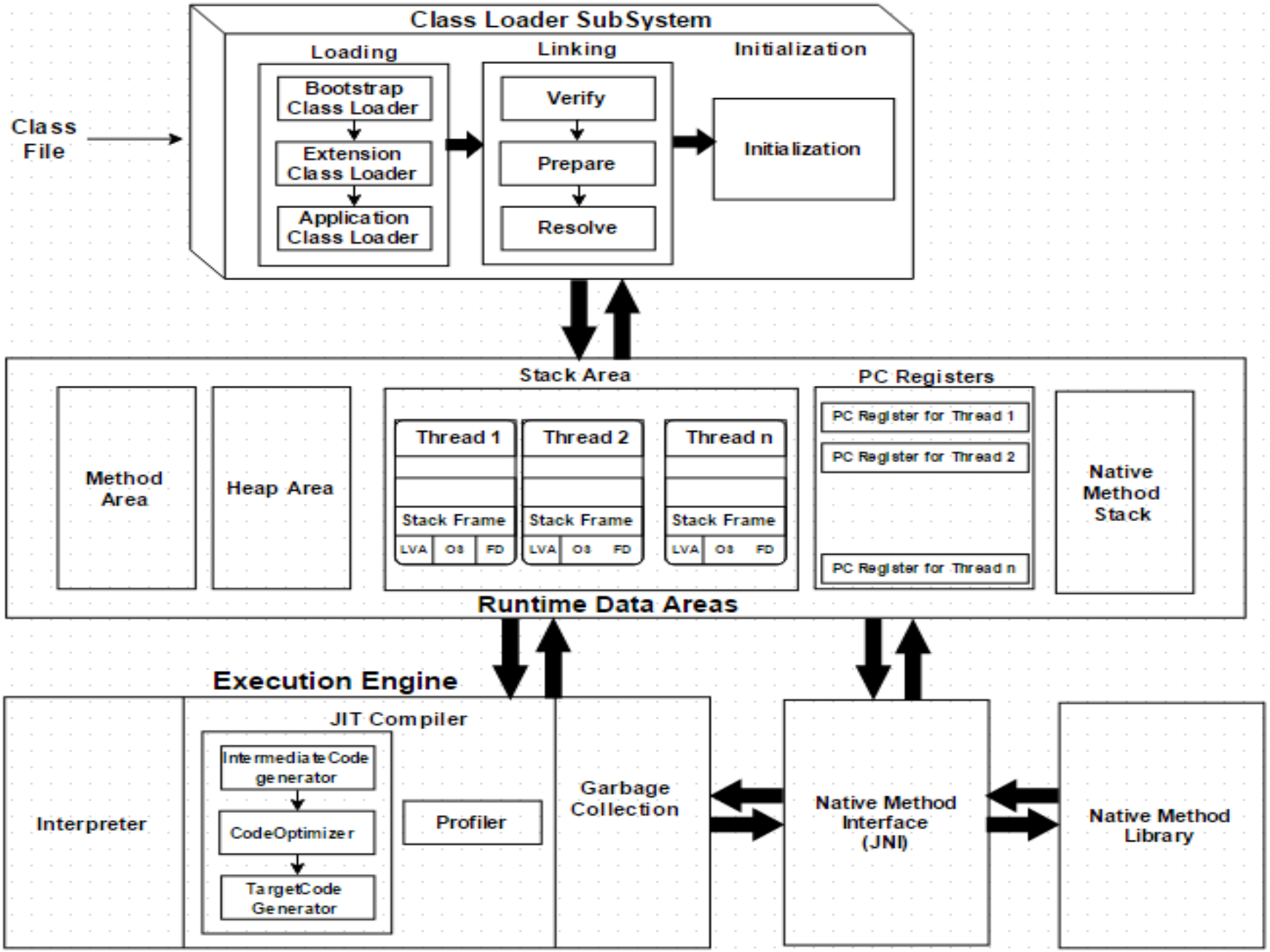
JRE = JVM + Libraries 3 **JDK = JRE + Development Tools**

Java Application

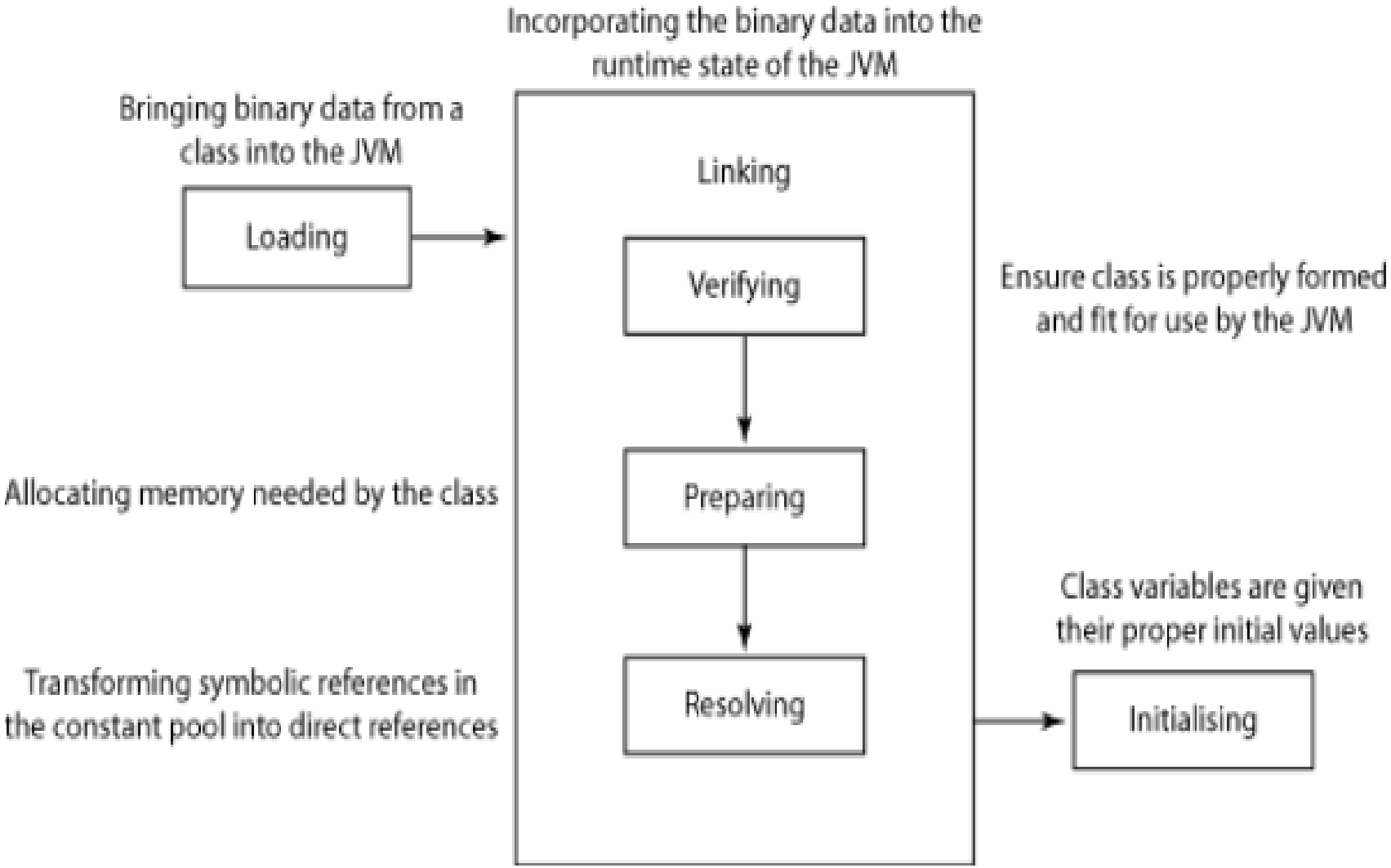
- Can be considered as a set of classes called user-defined classes.
- Java program, unlike one written in C or C++, isn't a single executable file, but instead is composed of many individual class files, each of which corresponds to a single Java class.
- Additionally, these class files are not loaded into memory all at once, but rather are loaded on demand, as needed by the program.
- **Classpath** parameter tells the Java Virtual Machine (JVM) where to look for these user-defined classes.
- Every object in a Java program is an instance of a class. Every class in a Java program is loaded by a classloader

4

JVM Architecture Diagram



Phases of Class Loading



Loading Phase

- Classes will be loaded by this component. BootStrap ClassLoader, Extension ClassLoader, Application ClassLoader are the three class loader which will help in achieving it.

- **BootStrap ClassLoader** – Responsible for loading classes from the bootstrap classpath, nothing but **rt.jar.** Highest priority will be given to this loader.
- **Extension ClassLoader** – Responsible for loading classes which are inside **ext** folder **(jre\lib)**
- **Application ClassLoader** – Responsible for loading **Application Level Classpath**, path mentioned Environment Variable etc.

- The above **Class Loaders** will follow **Delegation Hierarchy Algorithm** while loading the class files.

Linking Phase

- **ByteCode Verification** – Class loader does a number of checks on the bytecodes of the class to ensure that it is well formed and well behaved. This is detailed in JVM spec.

- **Class Preparation** – This stage prepares JVM necessary data structures that represent fields, methods, and implemented interfaces that are defined within each class.

- **Resolution** – All **symbolic memory references** are replaced with the **direct references** from **Method Area**. (previous step had built up the data structures with relative offsets and symbol references - in this step we "resolve“)

Initialization Phase

- This is the final phase of Class Loading, here all static variable will be assigned with the original user defined values and **static block** will be executed.
- <clinit> for a class is run. And your static fields of a class are finally set up to user defined values(if any).
- At the end of these three phases, a class is fully loaded and is ready for use.
- Note that class loading can be performed in a lazy manner and therefore some parts of the class loading process may be done on first use of the class rather than at load time.

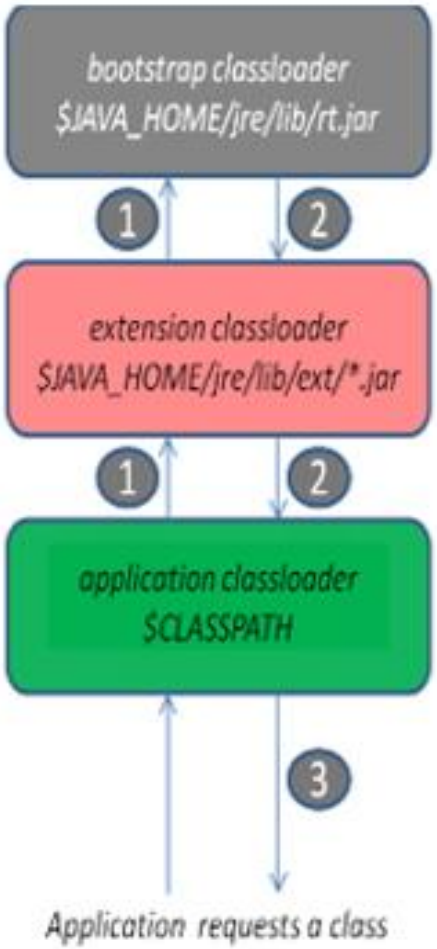
Principles of Java Class Loading

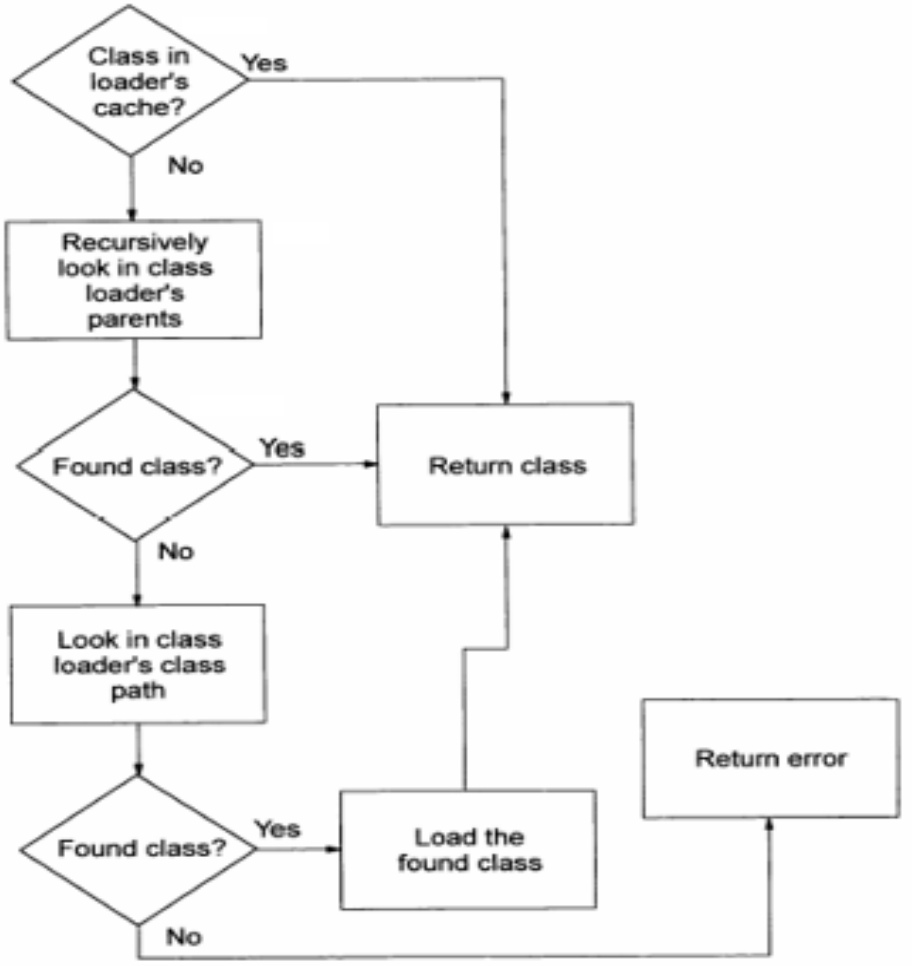
1. **Delegation** – The delegation principle guides us when class is not already loaded. In such a case, the child class loader asks its parent to load the class until bootstrap class loader is checked.
   - ✓Cache
   - ✓Parent
   - ✓Self

2. **Visibility**

3. **Uniqueness**

Delegation Model and Logic





Principles of Java Class Loading…

2. **Visibility** – The visibility principle ensures that the child class loader can see all the classes loaded by its parent. But, the inverse of that is not true, the parent can't see the class loaded by its child.

3. **Uniqueness** – The uniqueness principle ensure that a class will be loaded exactly once in the lifetime of a class loader hierarchy (*since the child has visibility to parent class, it never tries to load classes already loaded by the parent, but two siblings may end up loading the same class in their respective class loaders*)

Explicit Vs Implicit Loading

- **Explicit class loading** occurs when a class is loaded using one of the following method calls:
  - cl.loadClass() (where cl is an instance of java.lang.ClassLoader)
  - Class.forName() (the starting class loader is the defining class loader of the current class)

```
// Create a Class instance of the MazdaAxela class.
Class clazz = Class.forName("org.cars.MazdaAxela");

// Creates an instance of the Car.
Car mzAxela = (Car) clazz.newInstance();
```

- **Implicit class loading** occurs when a class is loaded as result of a reference, instantiation, or inheritance (not via an explicit method call). In each of these cases, the loading is initiated under the covers and the JVM resolves the necessary references and loads the class. As with explicit class loading, if the class is already loaded, then a reference is simply returned; otherwise, the loader goes through the delegation model to load the class.

- Classes are often loaded through a combination of explicit and implicit class loading. For example, a class loader could first load a class explicitly and then load all of its referenced classes implicitly.

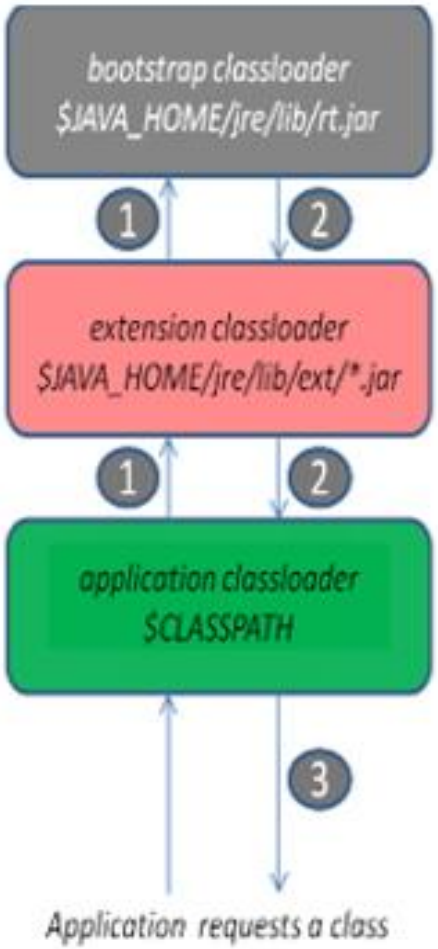
Types of Class Loaders

Initiating ClassLoader

- ClassLoader that received the initial request to the load the class

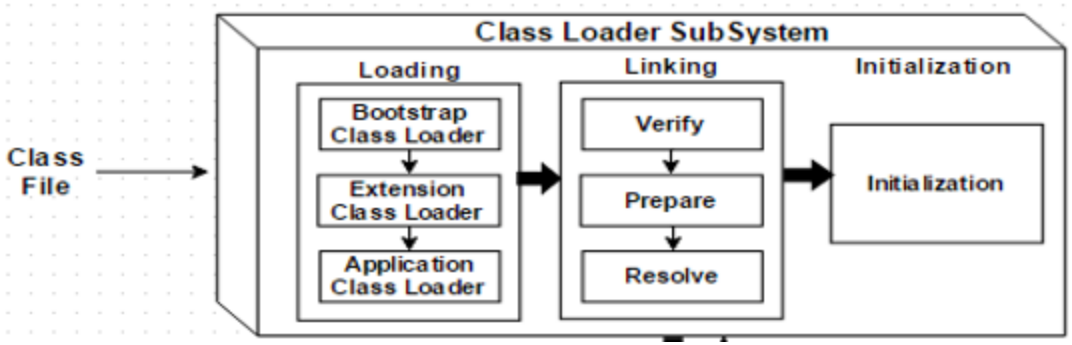
Defining ClassLoader

- ClassLoader that actually loads the class.



Key ClassLoading Exceptions

- ClassNotFoundException: Loading phase
- NoClassDefFoundError : Loading phase



- ClassFormatError : **Early** Verify stage of Linking Phase
- UnsatisfiedLinkError: Resolve stage of Linking Phase
- ClassCastException : Beyond Classloading

ClassNotFoundException

- **ClassNotFoundException** is the most common type of class loading exception. It occurs during the **loading phase.**

- Thrown when an application tries to load in a class through its string name using:
  - **forName**() method in class Class.
  - **loadClass**() method in class ClassLoader.

  but no definition for the class with the specified name could be found.

- So a ClassNotFoundException is thrown if an explicit attempt to load a class fails.

- By throwing a ClassNotFoundException, the class loader informs you that the bytecode required to define the class is simply **not present in the locations** where the class loader is looking for it.

```
public class ClassNotFoundExceptionTest {

   public static void main(String args[]) {
      try {
         URLClassLoader loader = new
URLClassLoader(new URL[] { new URL(

"file://C:/CL_Article/ClassNotFoundException/")});
         loader.loadClass("DoesNotExist");
      } catch (ClassNotFoundException e) {
         e.printStackTrace();
      } catch (MalformedURLException e) {
         e.printStackTrace();
      }
   }
}
```

NoClassDefFoundError

- NoClassDefFoundError is another common exception thrown by the class loader during the **loading phase.**
- The JVM specification defines NoClassDefFoundError as follows:
  - Thrown if the Java virtual machine or a ClassLoader instance tries to load in the definition of a class (as part of a normal method call or as part of creating a new instance using the new expression) and no definition of the class could be found.
- The searched-for class definition existed when the currently executing class was compiled, but the definition can no longer be found.
- Essentially, this means that a NoClassDefFoundError is thrown as a result of a **unsuccessful implicit class load.**
- Class existed at compile time, but no longer available
- Failure to load implicit dependent class or it has a bad format.

NoClassDefFoundErrorTest.java

```
public class NoClassDefFoundErrorTest {
    public static void main(String[] args) {
        A a = new A();
    }
}
```

A.java

```
public class A extends B {
}
```

B.java

```
public class B {
}
```

NoClassDefFoundError…

- Class A extends class B; so, when class A is being loaded, the class loader implicitly loads class B. Because class B is not present, a NoClassDefFoundError is thrown. If the class loader had been explicitly told to load class B (by a loadClass("B") call, for instance), a ClassNotFoundException would have been thrown instead.

- Situations like this can happen when classes are **missed during packaging or deployment.**

- In above example, A extends B; however, the same error would still occur if A referenced B in any other way -- as a method parameter, for example, or as an instance field. If the relationship between the two classes were one of reference rather than of inheritance, then the error would be thrown on the first active use of A, rather than during the loading of A.

ClassFormatError

- The JVM specification states that a ClassFormatError is thrown if:
  - The binary data that purports to specify a requested compiled class or interface is malformed.

- This exception is thrown during the verification stage of the linking phase of class loading. The binary data can be malformed if the bytecodes have been changed -- if the major or minor number has been changed, for instance. This could occur if the bytecodes had been deliberately hacked, for example, or if an error had occurred when transferring the class file over a network.

- Only way to fix this problem is to obtain a corrected copy of the bytecodes, possibly by recompiling.

UnsatisfiedLinkError

- The class loader plays an important role in linking a native call to its appropriate native definition.
- An UnsatisfiedLinkError occurs during the **resolving stage of the linking phase** when a program tries to load an absent or misplaced native library.
- The JVM specification says that an UnsatisfiedLinkError is:
  - Thrown if the Java Virtual Machine cannot find an appropriate native language definition of a method declared native.
- Native methods invoked via System.loadLibrary("myNativeLibrary");
- Loading of a native library is initiated by the class loader of the class that calls System.loadLibrary()

```
public class UnsatisfiedLinkErrorTest {

   public native void call_A_Native_Method();

   static {

System.loadLibrary("myNativeLibrary");
   }

   public static void main(String[] args) {
      new
UnsatisfiedLinkErrorTest().call_A_Native_Meth
od();
   }
}
```

ClassCastException

- Thrown as a result of incompatible types being found in a type comparison. The JVM specification says a that ClassCastException is:
  - Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.

- Occurs when the following conditions exists.
  - The source object is not an instance of the target class.
    Car myCar = (Car) myDog
  - The classloader that loaded the source class is different from the classloader that loaded the target class.
    Car myCar = (Car) yourCar

Debugging Class Loading Issues

- **-Dibm.cl.verbose=<class name>:** Detailed Information regarding where class loaders look for classes and which class loader loads a particular class.

```
ExtClassLoader attempting to find MyClass
ExtClassLoader using classpath [.....]
ExtClassLoader could not find MyClass.class in C:\Program%20Files\IBM\Java60\jre\lib\ext\dtfj.jar
[.....]
ExtClassLoader could not find MyClass

AppClassLoader attempting to find MyClass
AppClassLoader using classpath C:\Users\IBrijesh
AppClassLoader found MyClass.class in C:\Users\Brijesh
AppClassLoader found MyClass
```

- **-verbose:dynload:** The JVM option,-verbose:dynload provides detailed information as each class is loaded by the JVM, including class and package name for class files that were in jar file along with the name and directory path of the jar file. Details of the size of the class and the time taken to load the class is also written out to terminal.

```
<Loaded java/lang/Object from C:\Program Files\IBM\Java60\jre\lib\vm.jar>
<  Class size 1555; ROM size 1688; debug size 0>
<  Read time 67 usec; Load time 54 usec; Translate time 57 usec>
```

Debugging Class Loading Issues…

- **-verbose:class :** Displays information about classloader related events including ones from shared class cache.

- **Javacore:** Class loader section includes the information about defined class loaders and the relationship between them and also lists of classes loaded by each class loader

  Eg: ***-Xdump:java:events=throw,range=1..1,filter=java/lang/ClassNotFoundException***

```
------------------------------------------------------------------------
CLASSES subcomponent dump routine
================================
 Classloader summaries
        12345678: 1=primordial,2=extension,3=shareable,4=middleware,5=system,6=trusted,7=application,8=delegating
R       p---st-- Loader *System*(0x00007FF40A044F08)
IB              Number of loaded libraries 3
L               Number of loaded classes 451
R       -x--st-- Loader sun/misc/Launcher$ExtClassLoader(0x00007FF40A04BDC0), Parent *none*(0x0000000000000000)
IB              Number of loaded libraries 0
L               Number of loaded classes 0
R       -----ta- Loader sun/misc/Launcher$AppClassLoader(0x00007FF40A064798), Parent sun/misc/Launcher$ExtClassLoader(0x00007FF40A04BDC0)
IB              Number of loaded libraries 0
L               Number of loaded classes 1
 ClassLoader loaded libraries
        Loader *System*(0x00007FF40A044F08)
               /opt/ibm/biginsights/jdk/jre/lib/amd64/java
               /opt/ibm/biginsights/jdk/jre/lib/amd64/default/jclscar_24
               /opt/ibm/biginsights/jdk/jre/lib/amd64/zip
 ClassLoader loaded classes
        Loader *System*(0x00007FF40A044F08)
               sun/net/www/MessageHeader(0x00007FF430581550)
               java/io/FilePermission(0x00007FF430581F90)
               java/io/FilePermissionCollection(0x00007FF4305829B0)
               java/security/AllPermission(0x00007FF430583040)
               java/security/BasicPermissionCollection(0x00007FF4305833A0)
               org/apache/harmony/security/fortress/DefaultPolicy$ProtectionDomainCache(0x00007FF430583B80)
```

Debugging Class Loading Issues…

- **Javacore:** Class loader section includes the information about defined class loaders and the relationship between them and also lists of classes loaded by each class loader.

- Using the information provided in the Javadump, it is possible to ascertain which class loaders exist within the system. This includes any user-defined class loaders. From the lists of loaded classes, it is possible to find out which class loader loaded a particular class. If the class cannot be found, that means that it was not loaded by any of the class loaders present in the system (which would usually result in a ClassNotFoundException).

- Class loader namespace problems. A class loader namespace is a combination of a class loader and all the classes that it has loaded. For example, if a particular class is present but is loaded by the wrong class loader (sometimes resulting in a NoClassDefFoundError), then the namespace is incorrect -- that is, the class is on the wrong classpath. To rectify such problems, try putting the class in a different location -- in the normal Java classpath, for instance -- and make sure that it gets loaded by the system class loader.

Java Class Loader API

```
public abstract class ClassLoader extends Object {
  protected ClassLoader(ClassLoader parent);

  // Converts an array of bytes into an instance of class
  protected final Class defineClass(String name,byte[] b,int off,int len)
                                    throws ClassFormatError;

  // Finds the class with the specified binary name.
  protected Class findClass(String className) throws
                                    ClassNotFoundException;

  // Returns the class with the name, if it has already been loaded.
  protected final Class findLoadedClass(String name);


  // Loads the class with the specified binary name.
  public class loadClass(String className) throws ClassNotFoundException;
}
```

Creating a Custom Class Loader

```
public class loadClass(String name) throws ClassNotFoundException {
     // Check whether the class is already loaded
     Class c = findLoadedClass(name);
     if( c == null) {
          try {
               // Delegation happens here
               c.getParent().loadClass(name)
          } catch (ClassNotFoundException e) { // Ignored }

          if (c == null) {
               // Load the class from the local repositories
               c = findClass(name);
          }
     }
     return c;
}
```

Power of Custom Class Loaders

Here are a few ideas for interesting ClassLoaders:

- **Security**. Your ClassLoader could examine classes before they are handed off to the JVM to see if they have a proper digital signature. You can also create a kind of "sandbox" that disallows certain kinds of method calls by examining the source code and rejecting classes that try to do things outside the sandbox.

- **Encryption**. It's possible to create a ClassLoader that decrypts on the fly, so that your class files on disk are not readable by someone with a decompiler. The user must supply a password to run the program, and the password is used to decrypt the code.

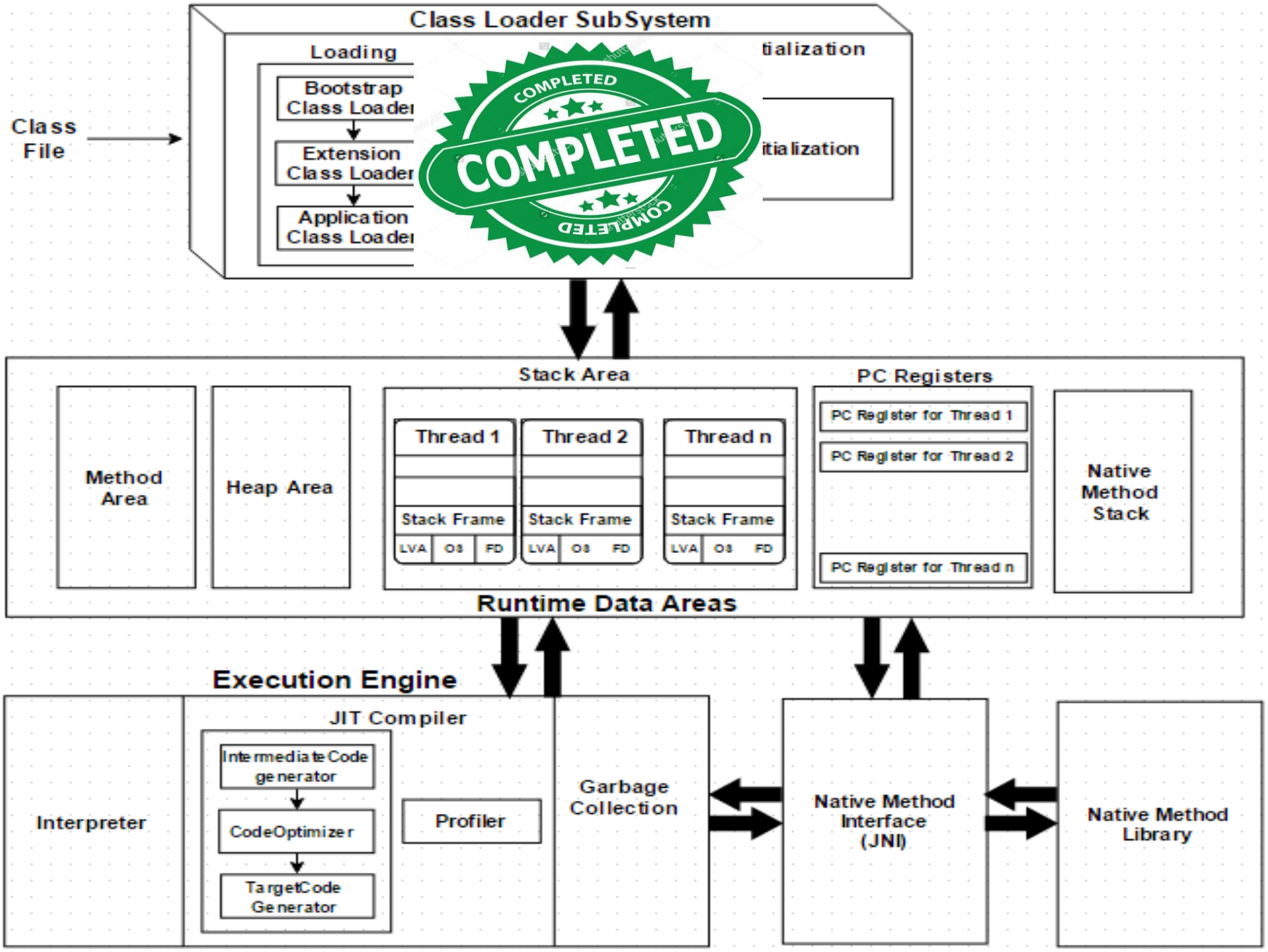
Power of Custom Class Loaders…

- **Archiving**. Want to distribute your code in a special format or with special compression? Your ClassLoader can pull raw class file bytes from any source it wants.

- **Self-extracting programs**. It's possible to compile an entire Java application into a single executable class file that contains compressed and/or encrypted class file data, along with an integral ClassLoader; when the program is run, it unpacks itself entirely in memory -- no need to install first.

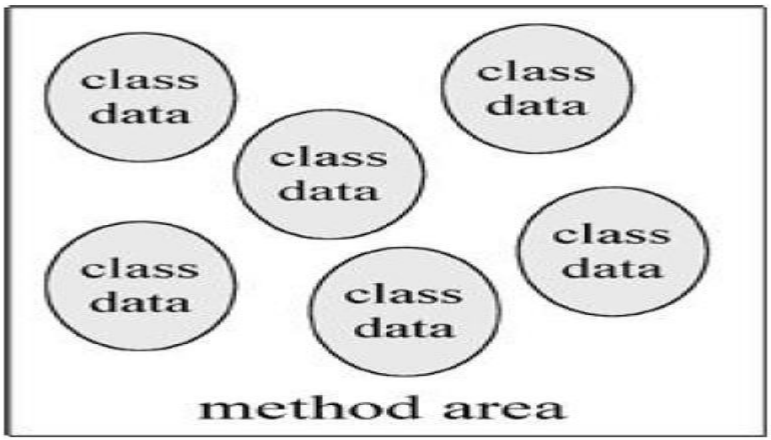
- **Dynamic generation**. They sky's the limit here. You can generate classes that refer to other classes that haven't been generated yet -- create entire classes on the fly and bring them into the JVM without missing a beat.

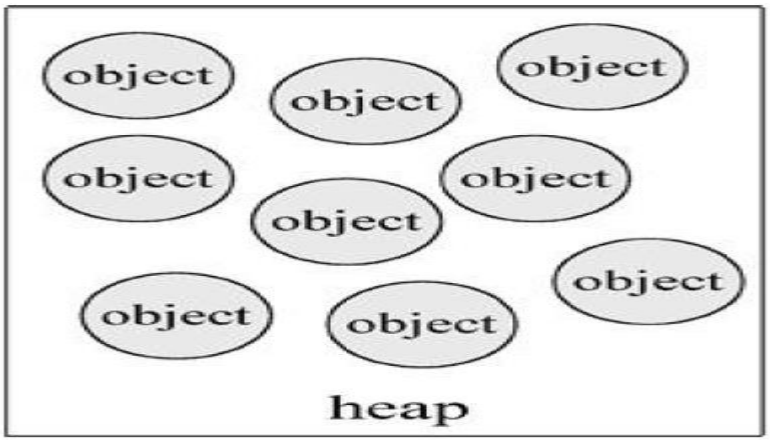
JVM Architecture Diagram



Runtime Data Area

- **Method Area** – All the Class level data will be stored here including static variables. Method Area is one per JVM and it is a shared resource.





- **Heap Area** – All the Objects and its corresponding instance variables and arrays will be stored here. Heap Area is also one per JVM since Method area and Heap area shares memory for multiple threads the data stored is not thread safe.

Runtime Data Area…

- **Stack Area** – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called as Stack Frame. All local variables will be created in the stack memory. Stack area is thread safe since it is not a shared resource. Stack Frame is divided into three sub-entities such as

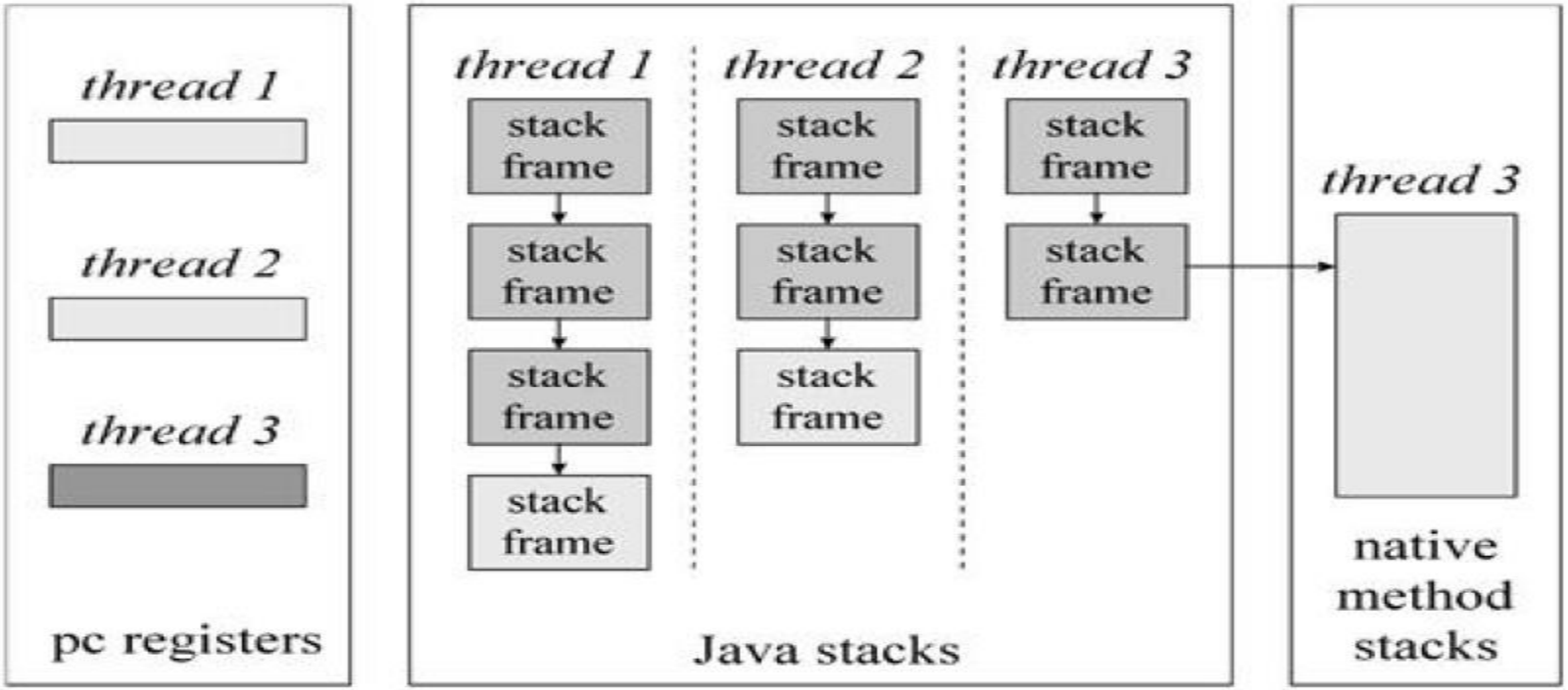
  - **Local Variable Array** – Related to the method how many local variables are involved and the corresponding values will be stored here.

  - **Operand stack** – If any intermediate operation is required to perform, operand stack act as runtime workspace to perform the operation.

  - **Frame data** – All symbols corresponding to the method is stored here. In the case of any exception, the catch block information will be maintained in the frame data.

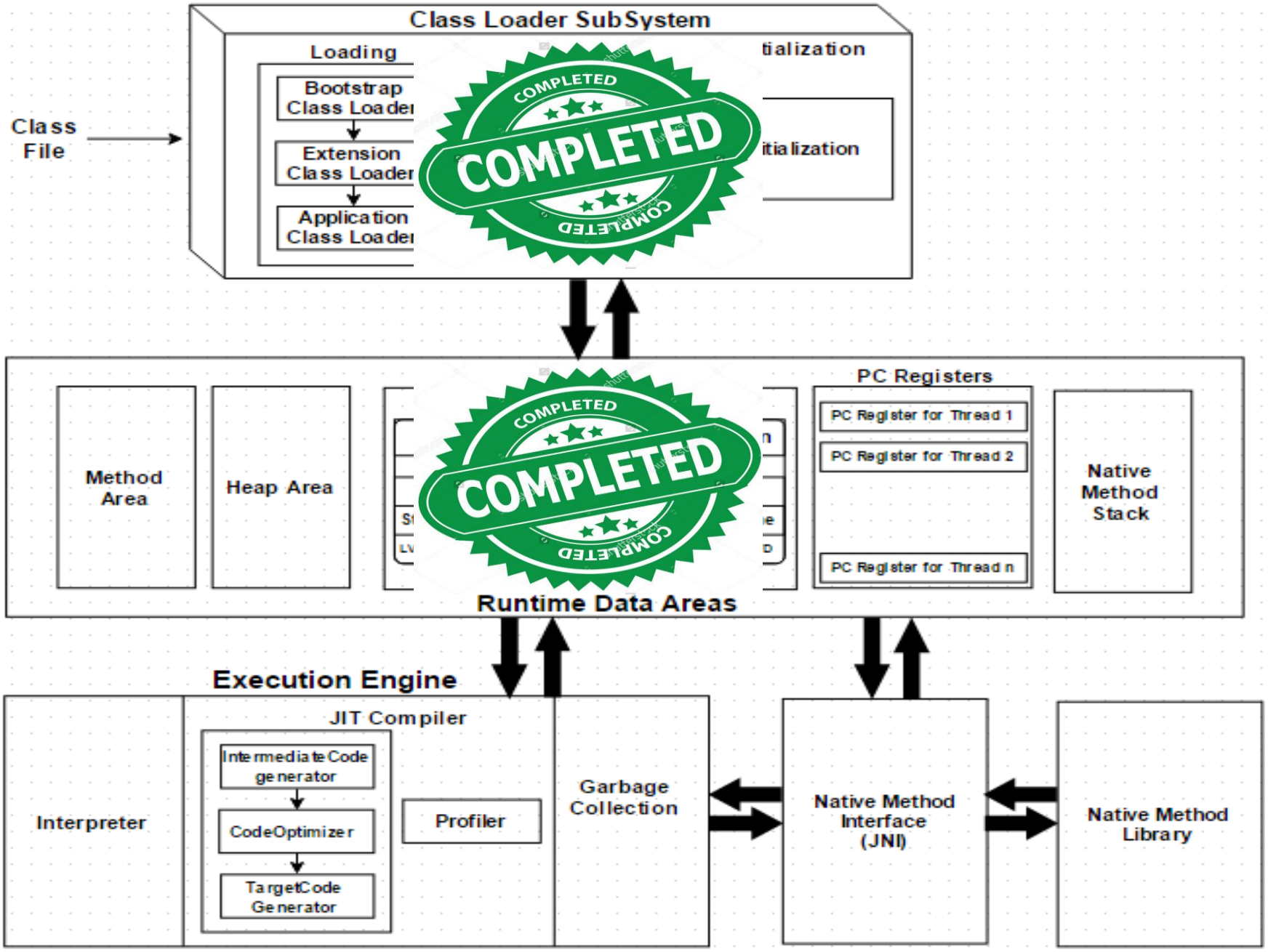
Runtime Data Area…

- **PC Registers** – Each thread will have separate **PC Registers,** to hold address of **current executing instruction** once the instruction is executed the PC register will be **updated** with the next instruction.



- **Native Method stacks** – Native Method Stack holds native method information. For every thread, separate native method stack will be created.

JVM Architecture Diagram



Execution Engine

- Bytecode which is assigned to the **Runtime Data Area** will be executed by the Execution Engine.

- Execution Engine reads the byte code and executes one by one.

- Execution Engine consists of
  - **Interpreter**
  - **JIT Compiler**
  - **Garbage Collector**

Interpreter

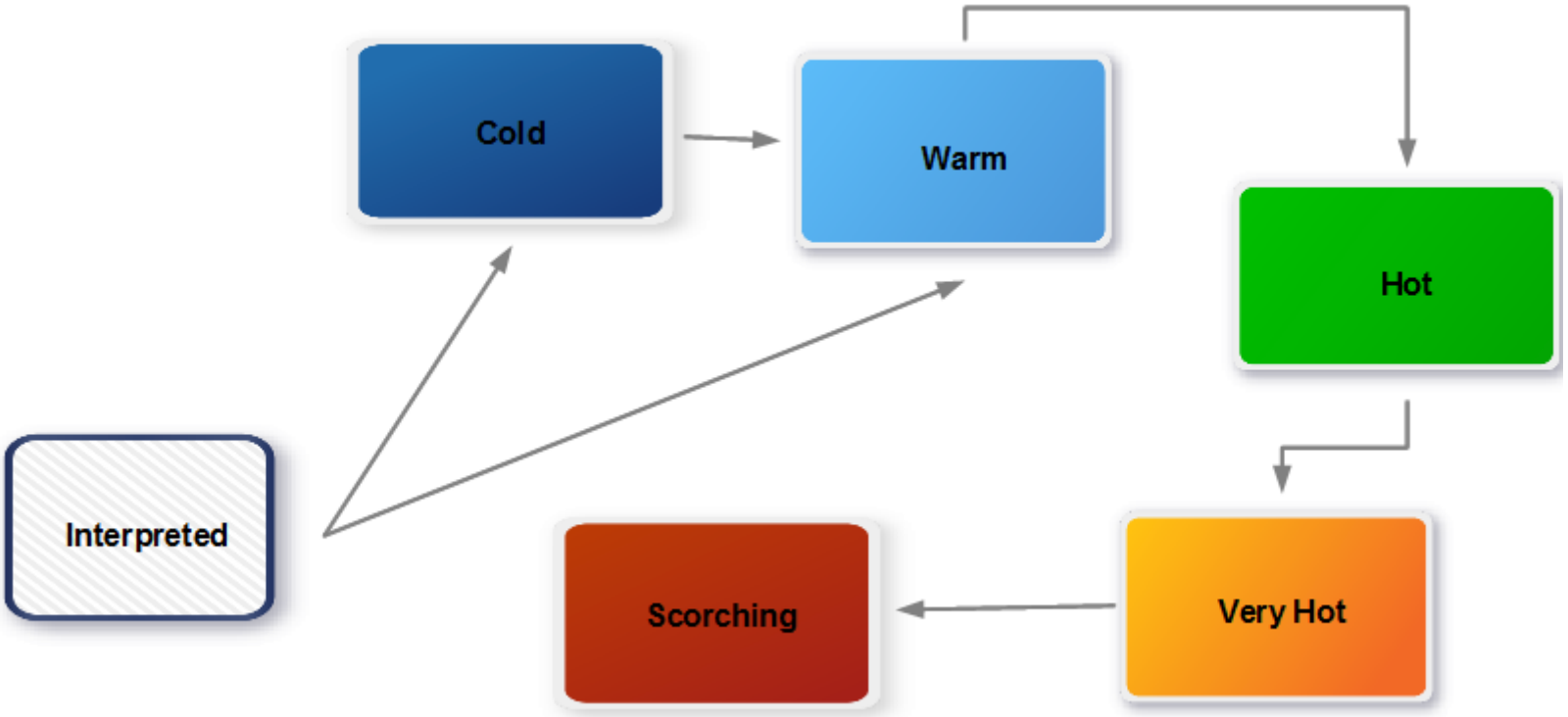
- **Interpreter** – Reads the bytecode, interprets it and executes it one by one.
- Interpreter interprets the bytecode faster but executes slowly.
- Disadvantage of the interpreter is that when one method called multiple times, every time interpretation is required.

JIT Compiler

- **JIT Compiler** helps us to overcome the disadvantage of the Interpreter (the single method is interpreted multiple times for multiple calls), The **Execution Engine** uses Interpreter to read and interprets the bytecode but when it came across repeated code it uses **JIT compiler** which compiles the entire Java bytecode once and changes it to native code. This native code will be used directly from next time onwards for repeated method calls.
- Methods gets compiled at run time.
- Dynamic observation of the execution of code via profiling to aggressively improve hot code.
- Interpreter profiling to adapt compilation to compiled methods for block reordering, loop unrolling, etc.
- Multiple optimization levels, multiple recompilations of the same method, many new optimizations.

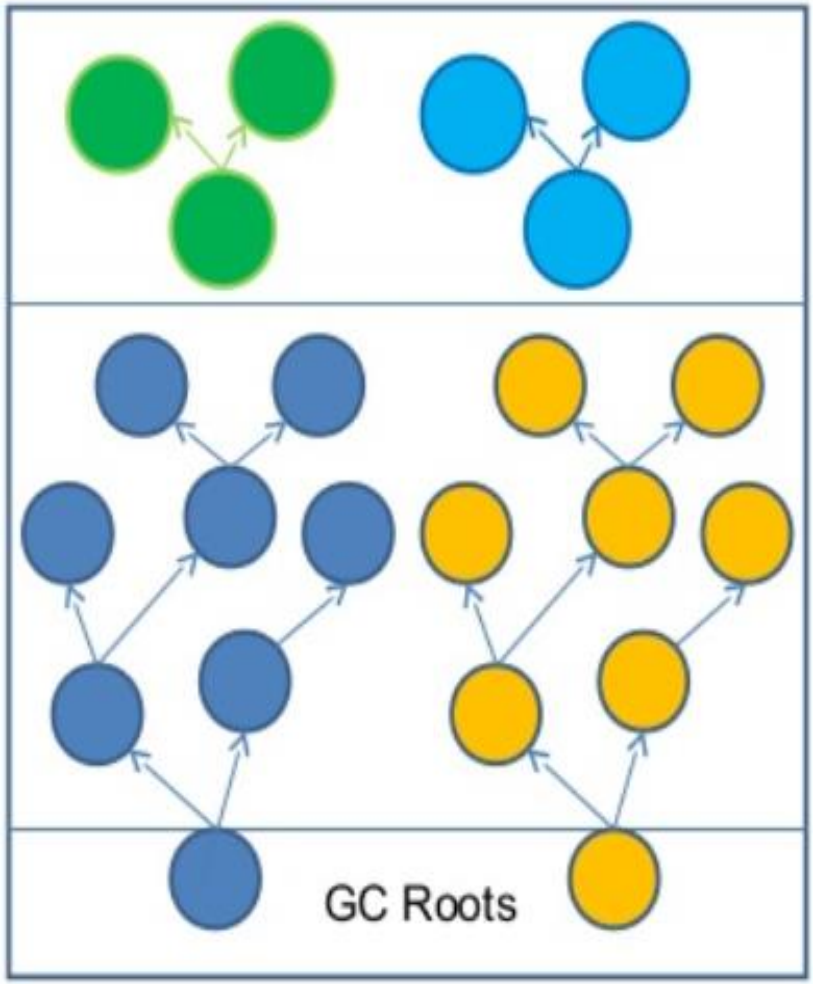
JIT Compiler Optimization levels

- Spending CPU on compiling a method is an investment.
- So, we invest incrementally through optimization levels & recompilations.
- As the method's prominence (frequency of use) increases, we keep recompiling it to higher levels.
- At the higher levels, we run the costlier optimization algorithms – better code.



Garbage Collector

- **Garbage Collector** : Responsible for allocation and freeing of Java objects, Array objects and Java classes.
- Automatically reclaim the memory used by objects that are no longer referenced by the running application. It may also move objects as the application runs to reduce heap fragmentation..
- Garbage Collection can be triggered by calling “System.gc()”, OOM, or when internal GC memory threshold is reached.
- GC roots are Class/Static variables, Stack Variables, JNI references, another live HEAP object etc

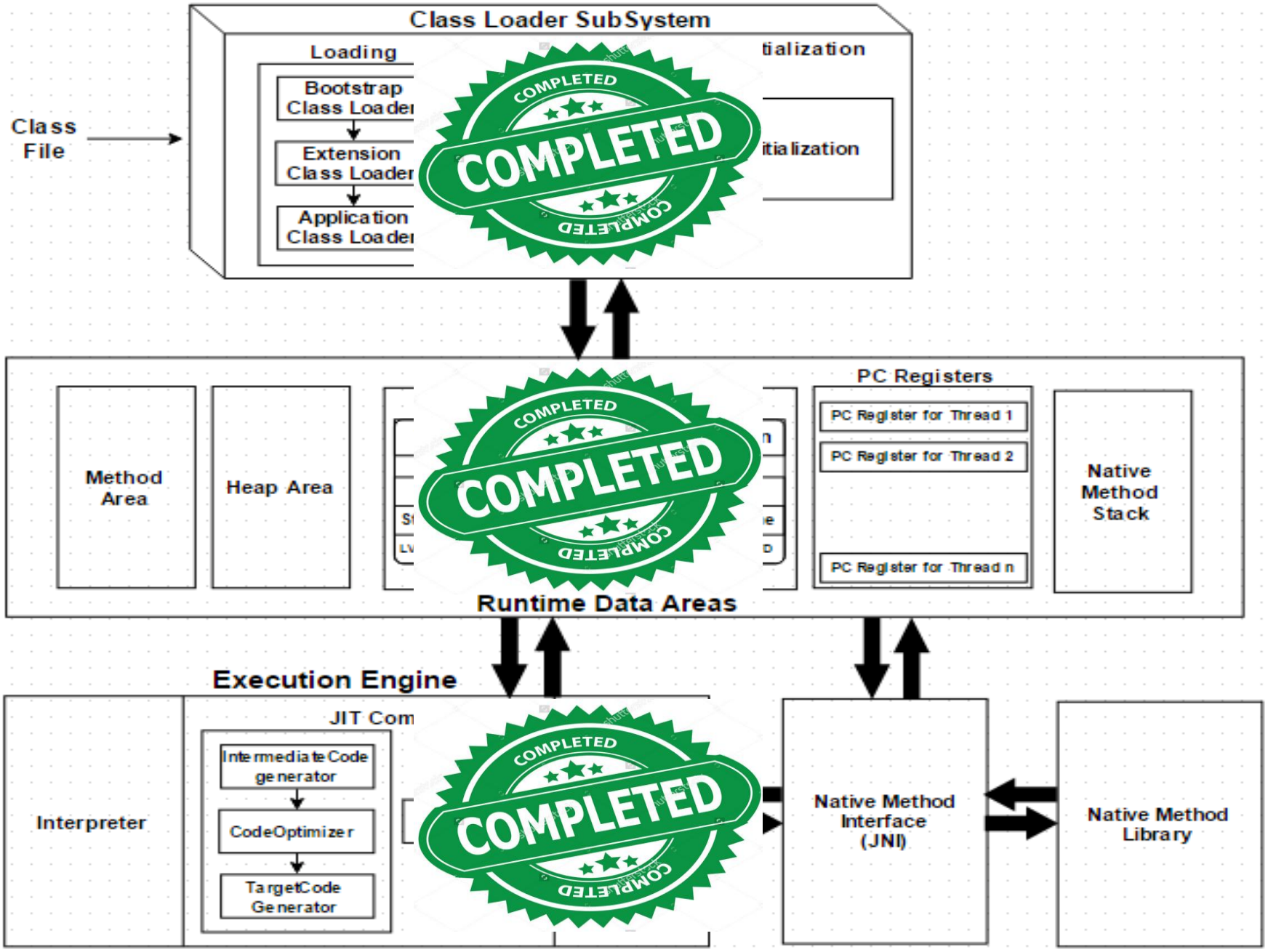


Execution Engine…

- Java Native Interface (JNI): JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

- Native Method Libraries : It is a Collection of the Native Libraries which is required for the Execution Engine.

JVM Architecture Diagram



References

- An introduction to class loading and debugging tools
- Understanding the Java ClassLoader
- Basic class loading exceptions
- Tackling more unusual class loading problems
- IBM JDK related Knowledge Center
- IBM JDK related forum

Questions ?